情報処理システム II ・第 12 回

2025 年 6 月 27 日

配布資料の内容

12.1 シェルによるプログラムの起動 . . . . . 12 – 1
12.2 シェルによるリダイレクション . . . . . 12 – 4
12.3 Unix 系 OS におけるプロセス管理関係の主なシステムコール . . . . . 12 – 5

12.1 シェルによるプログラムの起動

前回、Unix 系 OS (たとえば Linux) の環境では、シェルに対して

```
y240000@s01612h001:~$ cc -o myprog myprog.c
```

のように実行すべき外部コマンド (あるいは実行すべきプログラムのパス名) を入力すると、これをシェルが読み取って、指定されたプログラムを新たなプロセスとして起動することを説明しました。このときシェルが行なう手順は以下のようなものです。

1. `fork` というシステムコールを呼び出して、自分のプロセスの複製を作成する。
2. 複製によって生まれたシェルの新しいプロセスが、必要に応じて `open` 中のファイル記述子の入出力先を切り替える。
3. シェルの新しいプロセスが、`exceve` というシステムコールを呼び出して、指定したプログラムファイル (`/usr/bin/cc` など) を実行する。
4. 元のプロセスは、`wait` (あるいは `waitpid`) というシステムコールを呼び出して、新しいプロセスが終了するのを待つ。

メモ

fork システムコール　`fork` は、このシステムコールを呼び出したプロセスを複製して、新しいプロセスを生成します。新しく生成されたプロセスは、元のプロセスの子プロセスとなり、元のプロセスは新しいプロセスの親プロセスとなります。`open` 中のファイル記述子やカレントディレクトリ、環境変数の設定など、(プロセス ID を除く) 多くのプロセス管理情報 (プロセスの属性) が、元のプロセス (親プロセス) から、生成されたプロセス (子プロセス) に引き継がれます。プロセスの仮想アドレス空間は複製され、同じ内容を持つ独立した仮想アドレス空間となります。`fork` 直後では、同じ仮想アドレスの内容は、親プロセスも子プロセスも全く同じですが、その後、どちらかのプロセスがその内容を書き換えても、もう一方のプロセスの内容には影響を与えません。`open` しているファイル記述子やカレントディレクトリ、環境変数の設定などの属性についても同様です。

関数としての fork の戻り値は、子プロセスでは 0 となり、親プロセスでは、生成された子プロセスのプロセス ID[1] となります。プロセスの機械語プログラムを含めて仮想アドレス空間が複製されますので、関数 fork の呼び出しから戻った後、親も子も全く同じプログラムの実行を続けることになりますが、fork の戻り値が異なることを利用すると、次のプログラム例のように、親と子で違った処理をそれぞれ行なうことができます。

```
pid_t child_pid = fork();

if (child_pid == 0) {
    子プロセスが行なう処理
            ⋮
}
else {
    親プロセスが行なう処理
            ⋮
}
```

メモ

execve システムコール　シェルが新しいプログラムを起動する際には、まず fork を行なって子プロセスを生成し、上のプログラム例での「子プロセスが行なう処理」の部分で、例えば、

```
if (child_pid == 0) {
    char *argv[] = { "cc", "-o", "myprog", "myprog.c", NULL };
    char *envp[] = { "PATH=...", "LOGNAME=...", "LANG=...", ..., NULL };

    execve("/usr/bin/cc", argv, envp);
}
```

のように、execve システムコールを使って新しいプログラムを実行を開始します。exceve の第 1 引数が実行するプログラムファイルのパス名です。このシステムコールにより、このプロセス (元のシェルのプロセスの子プロセス) 仮想アドレス空間はすべて捨てられ、第 1 引数で指定されたプログラムファイル中に記録されている情報に基づき再構築されます[2]。

exceve の第 2 引数の argv は実行されるプログラムの引数となります。argv の各要素には、

```
argv[0] = "cc"
argv[1] = "-o"
argv[2] = "myprog"
argv[3] = "myprog.c"
argv[4] = NULL
```

[1]新しいプロセスの生成に (カーネルが) 失敗した場合は −1 が戻されます。

[2]第 4 回と第 7 回の配布資料を参照してください。

のように、シェルが標準入力から読み取ったコマンド行の文字列が設定され、最後に NULL (アドレスとしての 0) が置かれます。

C 言語からコンパイルされて得られたプログラムが起動される場合、この引数情報は、C ランタイムオブジェクト[3]から main 関数を呼び出す際の第 2 引数となります[4]。

```
int main(int argc, char *argv[]) {
    ⋮
}
```

exceve の第 3 引数 envp は、このプロセスの環境変数の新しい設定です。プロセスの環境変数の設定は envp で指定されたものに置き換えられます。仮想アドレス空間や引数、環境変数の設定などを除けば、プロセス管理情報 (プロセスの属性) は、execve の呼び出し前と呼び出し後で変更されることはありません。例えば、このプロセス (シェルの子プロセス) のプロセス ID や open 中のファイル記述子、カレントディレクトリなどはそのまま引き継がれます[5]。

指定したプログラムの起動に成功した場合、関数 execve を呼び出した機械語プログラムが置かれていた仮想アドレス空間は、新しいプログラムのものに置き換えられますので、関数 execve を呼び出したプログラムへ戻ってくることはありません。元のプログラムへ戻ってくるのは、execve の第 1 引数で指定したプログラムファイルの起動に失敗した場合のみとなります。

メモ

wait システムコール　プログラムを起動するために fork したシェルのプロセス (親プロセス) は、wait システムコールを呼び出して子プロセスが終了するの待ちます。wait を呼び出したプロセスは、自分の子プロセスのいずれかが終了するまでイベント待ち状態となります。

結局、シェルが (外部コマンドを含む) プログラムを起動する場合の全体の流れは、次のようなものとなります。

```
pid_t child_pid = fork();

if (child_pid == 0) {
    char *argv[] = { "cc", "-o" , "myprog", "myprog.c", NULL };
    char *envp[] = { "PATH=...", "LOGNAME=...", "LANG=...", ..., NULL };

    execve("/usr/bin/cc", argv, envp);
    fpritnf(stderr, "cc の起動に失敗しました\n");
```

[3]第 8 回の配布資料を参照してください。

[4]main 関数の第 1 引数 argc は、argv 配列に格納されている引数 (文字列) の数となります。

[5]ファイル記述子に関しては、fcntl というシステムコールであらかじめ設定しておくことで、execve 時に自動的に close することもできます。

```
        exit(1);
    }
    else {
        int stat;

        wait(&stat);
    }
```

12.2 シェルによるリダイレクション

fork は親プロセスが開いたファイル記述子を open された状態にしたまま子プロセスを生成し、execve も、そのままの状態で指定したプログラムを起動しますので、例えば、シェルに対して、

```
y240000@s01612h001:~$ ./myprog
```

のように入力して myprog を起動すると、図 1 のように myprog のプロセスの標準入力、標準出力、標準エラー出力はシェルと同じ疑似端末に接続された状態となります。

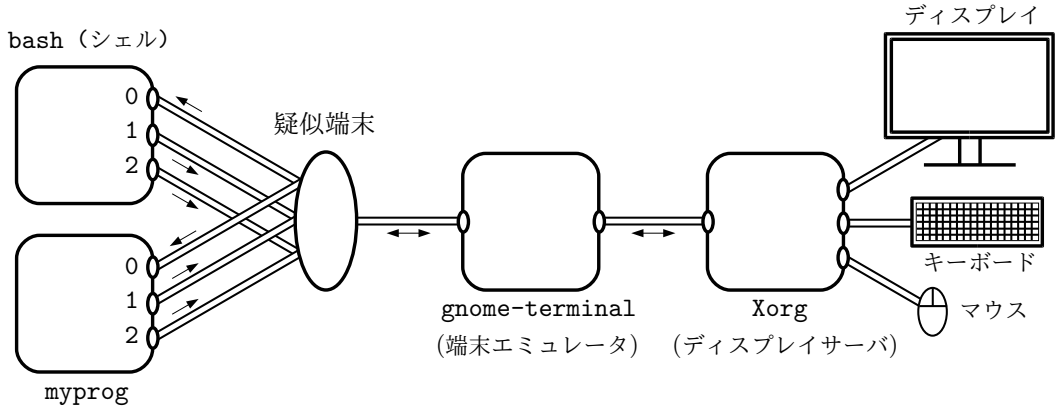

図 1: シェルと起動されたプログラム (myprog) と疑似端末との関係

このため、myprog が printf を呼び出すと、システムコール write(1, ...) が呼び出され、その出力は (疑似端末を介して) 端末エミュレータのウィンドウに表示されますし、myprog が呼び出した scanf は、システムコール read(0, ...) を呼び出し、端末エミュレータに対するキーボードからの入力を (疑似端末を介して) 受け取ることになります。

一方、例えば

```
y240000@s01612h001:~$ ./myprog <input.data >output.data
```

のように、標準入力や標準出力のリダイレクションを指定してプログラムを起動した場合、シェルは fork した後、子プロセスの方で、まず標準入力や標準出力を指定されたファイル (input.data や output.data) に接続し直し[6]てから execve("./myprog", ...) を呼び出します。こうすると、myprog が同じように printf や scanf を呼び出しても、(端末エミュレータにつながっている) 疑似端末の代わりに、これらのファイルが入出力先として使用されることになります。シェルによる

[6] close(0) に続いて、open("input.data", ..) を呼び出せば 0 番のファイル記述子 (標準入力) を input.data に接続し直すことができます。同様に、1 番 (標準出力) を output.data に接続し直すこともできます。

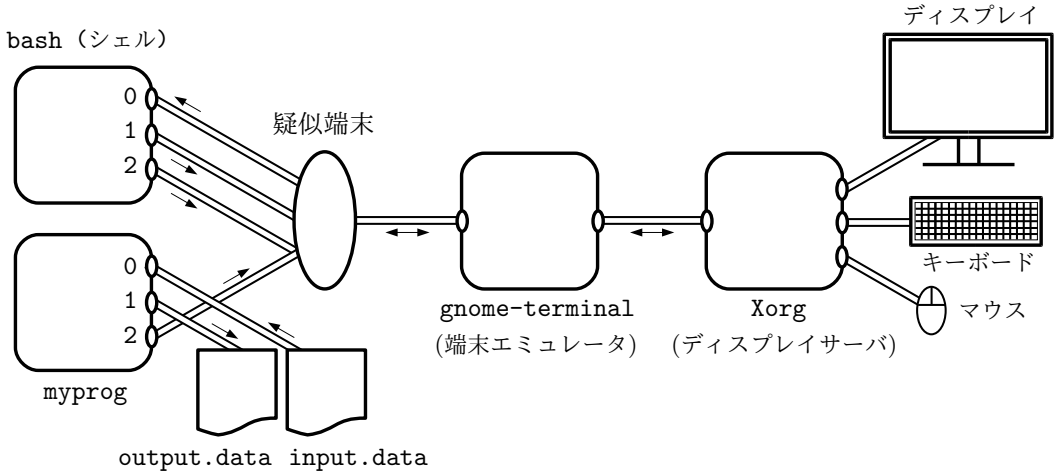

図 2: 標準入力と標準出力がリダイレクトされた myprog のプロセス

プログラム起動時の標準入力や標準出力のリダイレクションはこのような方法で実現されています (図 2)。

メモ

12.3 Unix 系 OS におけるプロセス管理関係の主なシステムコール

以下は、Linux などの Unix 系 OS におけるプロセス管理関連の主ななシステムコール群です。

`pid_t fork(void)`	現在のプロセスを複製して新しいプロセスを生成する。新しいプロセスは現在のプロセスの子プロセスとなり、現在のプロセスは新しいプロセスの親プロセスとなる。子プロセスでの戻り値は 0 となり、親プロセスでの戻り値は、子プロセスのプロセス ID となる。

`int execve(char *path,`
`  char *argv[],`
`  char *envp[])`
	`path` で指定したファイルを、`argv` を引数の設定、`envp` を環境変数の設定として実行する。実行中のプログラムの仮想メモリ空間は初期化される。オープン中のファイル記述子はそのまま保持される。プロセス ID は変更されない。

`void _exit(int c)`	終了コードを `c` として、現在のプロセスを直ちに終了する[7]。
`pid_t wait(int *stat)`	子プロセスが終了するのを待つ[8]。終了した子プロセスの終了コードが `*stat` に書き込まれ、プロセス ID が戻り値として戻される。

[7] C 言語の標準ライブラリ関数 `exit` は、このシステムコール (Linux では、その代りができる `exit_group` というシステムコール) を呼び出します。

[8] より正確には、終了するか、シグナルを受け取るのを待つ。

`pid_t waitpid(`
`  pid_t pid, int *stat,`
`  int opt)`
	プロセス ID が `pid` のプロセスが終了するのを待つ[8]。終了した子プロセスの終了コードが `*stat` に書き込まれ、プロセス ID が戻り値として戻される。引数 `opt` は待ち方の詳細を指定する。

`pid_t getpid(void)`	このプロセスのプロセス ID を戻す。
`pid_t getppid(void)`	このプロセスの親プロセス ID を戻す。
`int chdir(char *path)`	このプロセスのカレントディレクトリを `path` に設定する。

表中の関数プロトタイプに現れている `pid_t` という型は OS によって異なりますが、たとえば、情報実習室の Linux 環境 (64bit) では、これらは次のように定義されています。

```
typedef int pid_t;
```

情報処理システム II ・第 12 回・終わり